8-1991

# Explicit Clock Temporal Logic in Timing Constraints for Real-Time Systems

S. Ramanna

J. F. Peters III

SU-CIS-91-26

# *Explicit Clock Temporal Logic in Timing Constraints for Real-Time Systems*

S. Ramanna and J.F. Peters III

August 1991

*School of Computer and Information Science*
*Syracuse University*
*Suite 4-116, Center for Science and Technology*
*Syracuse, New York 13244-4100*

# Explicit Clock Temporal Logic in Timing Constraints for Real-Time Systems*

S. RAMANNA and J.F. PETERS III

*Syracuse University*
*School of Computer & Information Science*
*4-116 CST, Syracuse, NY 13244-4100 USA*

**Abstract.** A form of explicit clock temporal logic (called $TL_{rt}$) useful in specifying timing constraints on controller actions, a real-time database (rtdb) items, and constraints in a real-time constraint base (rtcb), is presented. Timing as well as other forms of constraints are stored in the rtcb. A knowledge-based approach to ensure the integrity of information in an rtdb is given. The rtcb is realized as a logic program called Constrainer, which is a historyless integrity checker for a real-time database. The consistency and integrity issues for an rtcb and rtdb are investigated. The formal bases for a temporally complete rtdb and knowledgeably complete controller are presented. A partial $TL_{rt}$ specification of a knowledgeable controller for a Gas Burner is given. An illustration of a rtdb and rtcb in the context of the sample real-time system is also given.

**Keywords.** Artificial intelligence; computer control; constraint theory; delays; monitoring; safety; system integrity.

## INTRODUCTION

Considerable work has been done on describing the behavior of hard, real-time systems (Alur, 1990; Cronhjort, 1988; Harel, 1990; Hankley, 1990a, 1990b; Henzinger, 1991a, 1991b; Kopetz, 1988, 1989a, 1989b; Ostroff, 1989, 1990; Peters, 1990a, 1990b, 1991a, 1991b, 1991c, 1991d; Ramanna 1991). A *hard, real-time system* (rts) is a computer system where the validity of results produced by the rts depend on both logical correctness and timeliness (Kopetz, 1989a). An rts consists of two main parts: controller and plant. The controller is a computer which processes input from the environment as well as plant and supplies control information to the plant ( hardware ). The controller relies on

* Research supported in part by the School of Computer and Information Science, Syracuse University, Syracuse, NY 13244-4100 USA, by the Research & Development Laboratories, Culver City, CA 90230-6608 USA. To appear in the IFAC AIRTC'91 Proceedings, Sept. 1991.

information provided by a real-time database to carry out its control functions. Informally, a *real-time database* (rtdb) is a collection of data items needed for instantaneous control, operator display, alarm monitoring, and other real-time system applications and which are invalidated by the passage of real-time (Kopetz, 1989a). The focus of this article is on a knowledge-based approach to ensure the integrity of the information in a real-time database. A knowledge-base is a database of facts about some application domain; these facts are refined by the acquisition of knowledge (Levesque, 1981). A knowledge-base can be realized as a logic program (Guessoum, 1990; Ramanna, 1990a, 1990b). In the context of real-time databases, a knowledge base containing constraints on items in the rtdb, is realized as a logic program called Constrainer. The term *knowledge base* is used synonymously with *constraint base* in this article. Each data item of the rtdb is a set of (attribute, value) pairs such that

data item = { (attribute, value) | attribute : string; value : pending }

The first pair (attribute, value) in the set serves as a key (or criterion) in identifying a data item. The term pending that the value of an attribute is context sensitive. Let $x \in$ rtdb, $(A_X, A_X\text{'value}) \in x$, where $A_X$ is an attribute of data item x and $A_X$'value is the value of the attribute. Let $T_{AX}$ (point of observation) be the reading of the external clock when $(A_X, A_X\text{'value})$ is first entered into the rtdb; let k be the number of ticks (beyond $T_{AX}$) during which $(A_X, A_X\text{'value})$ is valid. Then $T_{AX} + k$ (point of validity) is the upper limit on the validity of the information which $(A_X, A_X\text{'value})$ represents. Let q.time (t_use) be the time in state q of a controller when $(A_X, A_X\text{'value})$ is used. Then a *current observation* is one which is made at time t_use where $T_{AX} < t\_use < T_{AX} + k$. The notion of current observation provides the basis for a formal definition of a rtdb (Kopetz, 1989b).

Def. 0.1$^{o}$ A *real-time database* is a set of data items each of whose (attribute, value) pairs can serve as current observations needed for instantaneous control of a real-time system.

Def. 0.2$^{o}$ A rtdb is *temporally complete* if every $(A_X, A_X\text{'value})$ pair of every $x \in$ rtdb has a timing constraint.

A *timing constraint* is a predicate which specifies the duration over which a ($A_x$, $A_x$'value) pair of x $\epsilon$ rtdb is valid. The "knowledge" in rtcb is in terms of particular real-time applications, which provides a basis for computing the value of k (the point of validity); in this paper, we do not address the issue how the rtcb can be used to compute k. This issue is addressed in (Peters, 1991b; Ramanna, 1991). In this paper, we are interested in establishing a formal basis for a knowledge-based framework for guaranteeing the integrity of items in a temporally complete rtdb.

## EXPLICIT CLOCK TEMPORAL LOGIC

The behavior of a real-time system can be specified with Real-Time Temporal Logic (RTTL) given in (Ostroff, 1989, Harel 1990, Henzinger, 1991). RTTL is an explicit clock logic which uses data variables to reference an external clock in assertions. When temporal logic is applied to the study of processes, the formulas of temporal logic are interpreted as predicates over sequences of process states (Alpern, 1986). Each state occurs at some instant in time in which the values of process variables can be inspected. During a succession of states, changing values of state variables may entail changing truth values of predicates about state variables. Temporal formulas can be used to enumerate state transitions (transformations of one state into a new state) in a behavior as well as the order in which transitions are made.

RTTL provides a concise means of prescribing a property of a behavior of a rts controller or plant; such prescriptions are assertional. RTTL also provides a means of specifying constraints on values of data items over a sequence of rtdb states or timing restrictions on rtcb constraints. This form of temporal logic is essentially the same as the original temporal logic introduced by Manna and Pnueli (1981, 1983) with the addition of data variables such as T (for timing constraints) suggested by (Harel 1990, Henzinger, 1991). Except for some additional derived temporal operators taken from (Peters, 1990a), the temporal logic used in this article is the same as RTTL. For simplicity, we limit the presentation of RTTL to a discussion of the U (until) and temporal operators derived from U. We also introduce the derived temporal operators before, $\Diamond^{\omega}$ (infinitely often), and seq($p_1$, $p_2$, $p_3$,..., $p_n$) (a temporally quantified sequence of state predicates where $p_1$ holds before $p_2$, which holds before $p_3$, ..., before $p_n$). For the subset of RTTL (named $TL_{rt}$) we have chosen, the temporal language $TL_{rt}$ is defined as follows:

Alphabet

- A denumerable set of variables: x, ...
- A denumerable set of n-ary functions: f, g, ...
- A denumerable set of n-ary predicate symbols: p, q, ...
- Symbols ¬, or, ∀, (, ), U

Well-formed formulas of $TL_{rt}$ have the following syntax:

- Every atomic formula is a formula.
- If x is a variable and A is formula, then ∀ x A is a formula.
- If A and B are formulas, then ¬ A, (A or B), (A U B) are formulas.

*Semantics of Temporal Operators*

The ¬ (not), or, and ∀ (all) symbols have the usual semantics. In addition, the implication symbol ⟹ (i.e., p ⟹ q ≡ ¬ p or q) is used. In defining the following semantics, the notation

$$(q_0,...,q_x) \models p \text{ for } x >= 0$$

asserts that each of the states in the sequence $(q_0,...,q_x)$ satisfy predicate p. In what follows, let $q_0$ represent the current state in a behavior. Let p, q be first-order predicates. The semantics of U as well as the operators derived from U are as follows:

$p\ U\ q \equiv \exists\ k, x: 0 <= x <= k: (q_0,...,q_x) \models p$ and $q_k \models q$

p before q ≡ ∃ k: 1 <= k: $q_0 \models p$ and $(q_1,...,q_k) \models p\ U\ q$

$\Diamond p \equiv$ true U p

$q_k \models seq(p) \equiv q_k \models p$

$seq(p_1, (seq(p_2, ...,p_n))) \equiv p1$ before $seq(p_2, (seq(p_3,...,p_n))$

$\Diamond^{\omega} p \equiv seq(p, \Diamond^{\omega} p)$

$\Box p \equiv \neg \Diamond \neg p$

The predicate 'p U q' asserts that the predicate q eventually holds (either in the current or in some future state) and that the predicate p holds in the current state and in each of the states *until* the state when q holds. By contrast, 'p before q'

asserts that p is guaranteed to hold initially and sometime later q will hold. For this reason, *before* is called a precedence operator (Kröger, 1985). These powerful temporal operators provide the basis for the semantics of the remaining operators in the above list. $TL_{rt}$ is used to specify timing constraints on controller actions, rtdb data items, and rtcb constraints. The syntax for a well-formed constraint (wfc) is given in Appendix A.

## EXTERNAL CLOCKS AND TIMED BEHAVIORS

Timing constraints on items in a real-time database reference ticks of an external clock (denoted by Clock). Two types of variables are used to formulate constraints on rtdb items: rigid and flexible variables (Henzinger, 1991). A rigid variable r can be assigned a value in a particular rts state and r retains its value across state changes. By contrast, a flexible variable value can change with state changes. For example, the rigid variable T records the Clock value. We assume that the value of T can be changed when needed (this is analogous to resetting the external clock in a timed Büchi Automaton (Alur, 1991)). The flexible variable *time* gives the value of Clock in the current state. Clock readings are non-negative, real numbers. Each time an event occurs, a reading of Clock is associated with that event.

### *Semantics of Delay*

Responsiveness of a system is measured in terms of actual values of delays. The duration predicate delay(k) asserts that the external clock is allowed to run for k ticks before a timeout occurs. Delay(k) can be used to specify a lower bound on the number of ticks before an action is performed; delay(k) can also be used to specify an upper bound on the duration of an action, duration of validity of a constraint in an rtcb or the value of an attribute of a data item in an rtdb. A similar technique for specifying timing constraints on actions is used by Handelman and Stengel (1988).

### *Lower Time Bound on System Actions*

We can express a lower bound on the number of ticks before a system action begins. If we let ACT be the action to be performed in state q. We can express the fact that we let the external clock run for k ticks before performing ACT by writing informally "delay(k) before ACT." To see this, let T record the time in state q.

Assume action ACT is performed in state q. Written by itself, "ACT" is shorthand for the assertion "the action ACT is performed." Let sat(q | (q'), P) mean that predicate P is satisfied in state q of the state sequence (q, q'), and sat(q', Q) mean that predicate Q is satisfied in state q'. The double turnstile ⊨ reads "forces" or "satisfies." Then satisfaction of "delay(k) before ACT" over a state sequence (q, q') is expressed in clausal form as follows:

```
sat(q | (q'), delay(k) before ACT) :-
  q ⊨ delay(k) and T <= time < T + k,
  q' ⊨ ACT and time = T + k.
```

This says that the duration predicate is satisfied in state q and k ticks later the predicate ACT is satisfied in state q' .

*Upper Bound on Constraints, Attributes & Actions*

To specify an upper bound of k ticks on the timeliness of a constraint Pred in the rtcb, we write delay(k) Pred (or simply k Pred). To specify the upper bound of k ticks on the value of an attribute A in an rtdb, we write IsValid(A, k) within a constraint in the rtcb; this a timing constraint on an rtdb attribute. We can also express an upper bound on the number of ticks during a system action using delay(k). This is expressed rather simply by writing "ACT; delay(k)," which asserts that ACT cannot be continuously enabled for more than k ticks of the external clock. The predicate *timeout* is an enabling condition, which evaluates to true at the kth tick of the clock (i.e., an action which must be performed within k ticks times out, and a transition to the next state occurs). The meaning of this upper bound constraint can be explained concisely by using the satisfaction clause sat(q, P). Then the upper bound timing constraint can be defined as follows:

```
sat(q, ACT; delay(k))   :-
  q ⊨ ACT,
  q ⊨  time < T + k;              /* ; reads "or" */
  q ⊨  time = T + k and timeout.
```

## A CONSTRAINER FOR A REAL-TIME DATABASE

The constraint base rtcb maintains knowledge about a controller application. In this paper, we have restricted this knowledge to timing constraints. The rtcb is realized as a logic program called Constrainer, which determines the admissibility of a controller operation requiring a data item in a real-time database. A controller hardware interacts with the Constrainer to determine the integrity of needed rtdb control information. A Yes/No response from the Constrainer is contingent upon its subrules reporting success/failure of their subfunctions. A No answer prompts the Constrainer to declare the usage of a data item by the controller as inadmissible. Let q be a state of the Constrainer; let q | tail be a sequence of Constrainer states with q being the current state, tail being futures states, and op the name of an operation. Let rtcb be the constraint base managed by the Constrainer. The runtime system for the Constrainer maintains a state record used to store state variable values. This Constrainer state record has the following form:

```
type state record is
  time: real;                    /* current clock time */
  Pred: string;
  other: OtherType;
end record;
```

The constraints processed by the Constrainer are $TL_{rt}$ formulas. The Constrainer is given in clausal form in Fig. 1.

```
Constrainer(q | tail, Ask, attrib, , ) :-
  Constrainer(q | tail, Verify, attrib, , ).
  Constrainer (tail, op, , , ) .
Constrainer(q | tail, Tell, criterion, , constraint) :-
  Extract( constraint, delay, criterion, formula ) ,
  Accept(delay, criterion, constraint, q.time, q.db ) ,
  Constrainer ( tail, op, , , ) .
Constrainer( q | tail, Add, attrib, value, ) :-
  IsAdmissible (Add, attrib, q.cb, value ),          /* Rule Is.1 */
  Constrainer ( tail, op, , , ) .
Constrainer( q | tail, Update, attrib, value, ) :-
  IsAdmissible ( Update, attrib, q.cb, value ),
  Constrainer ( tail, op, , , ) .
Constrainer( q | tail, Delete, attrib, , ) :-
  IsAdmissible ( Delete, attrib, q.cb, ),
  Constrainer ( tail, op, , , ) .
Constrainer( q | tail, Verify, attrib, , ) :-
```

```
  IsAdmissible ( Verify, attrib, q.cb, ) ,
  Constrainer ( tail, op, , , ) .
Constrainer( q | tail, Read, attrib, value, )  :-
  Constrainer(q | tail, Verify, attrib, , ).
  Constrainer ( tail, op, , , ) .
```

Fig. 1 Constrainer Program

The Constrainer is used in the interpretation of constraints on data items which a controller wishes to utilize. The Constrainer rules take on a functional form where parameters from one rule can be passed on to the others, which is a deviation from pure Prolog. The Constrainer is defined in terms of four types of operations: Ask, Tell, Verify, and real-time database operations ( Add, Update, Delete, Read ). The Accept subrule for the Tell operation of the Constrainer is needed when we add a constraint to the constraint base. The acceptance of a Tell operation on a rtcb requires a check on (i) whether the duration on a new constraint is valid (it uses the current clock time, which it compares with a previous reading of the clock used in establishing a timing constraint on a knowledge sentence) [AcceptDuration]; (ii) whether the data items referenced by the new constraint exist in the rtdb [AcceptCriterion]; and (iii) whether the constraint itself is refuted by other constraints in the constraintbase [AcceptConstraint].

```
Accept ( delay,  criterion,  know, q.time, q.db ) :-          /* Rule A.1 */
   AcceptDuration ( delay , q.time ),
   AcceptCriterion ( criterion, q.db ) ,
   AcceptConstraint ( criterion, know,  q.time ).
AcceptDuration ( delay,  q.time ) :-
   q.time <= T + delay.                                  /* T is rigid variable */
AcceptCriterion ( criterion, db )  :-
  CollectItems ( criterion, itemlist, db ),
   member ( itemlist, db ).
AcceptConstraint (criterion, know,q.time ):-
   FindSentence( criterion, q.cb, knowset ) ,
   Decompose ( q | tail, know),
   IsConsistent ( knowset, know, q.time,  q.Pred ).          /* Rule IsC */
```

For a constraint on a data item to be accepted, one must ensure that the constraint is not refuted by any knowledge sentence in the knowledge set associated with the data item. In AcceptConstraint, the set of all knowledge sentences about the same items are returned by the FindSentence rule. The refutation mechanism of AcceptConstraint is expressed in IsConsistent ( knowset, know, q.time. q.Pred ). By contrast, the IsAdmissible (see Rule Is.1) subrule for Add, Update, Delete, Read, and

Verify of the Constrainer determines the admissibility of an operation on the items in the rtdb.

```
IsAdmissible ( op, attrib, q.cb, value ) :-
  Find( attrib, q.cb, knowset ) ,
  Check ( attrib, know | knowset, value ) .
```

where

```
Find (attrib | attribset, know | q.cb, knowset ):-
  ExtractCriterion (know,   criterion ),
  Match ( criterion, attrib ) ,
  Find (attribset, q.cb, know | knowset );          /* match found */
  Find ( attribset, q.cb, knowset ).                /* match  not found */

Match (" X " | criterion,  X ) .
Match (" Y " | criterion,  X ) :-
  Match (criterion,  X ).

Check( attrib, know | knowset, value ) :-           /* Rule C.1 */
  Extract( know, delay, criterion, formula ) ,
  Accept ( delay,  criterion,  know, q.time,  q.db ),
  Admissible( q, op, attrib, formula,value ) ,
  Check (  attrib, knowset, value ) .
```

The subrule

```
Find ( attrib | attribset, know | q.cb,  knowset )
```

obtains all the constraints associated with the data items of the operation. The subrule

```
Check ( attrib, know | knowset, value )
```

determines (i) whether the constraints are valid with respect to the constraint base, (ii) whether the substitution of values in each constraint associated with the items results in a valid formula. The subrule

```
Admissible(q,op,attrib,  formula,  value) :-
  Substitute (attrib,  formula,  value,  eval).
```

deals with the admissibility of an operation on the data items, by grounding the variables of a constraint with the new data values that are provided by the operation. The Extract, FindSentence, IsConsistent, ExtractCriterion, and Admissible subrules are given in the Appendix B.

*Note on Acceptance of Timing Constraint*

The AcceptDuration subrule (see Rule A.1) checks the validity of the duration of timing constraints. This rule "accepts" a timing constraint with upper bound k, provided that the current time given by q.time is within k ticks of the external clock (this is measured by comparing q.time with T + k, where T is a rigid variable which stores the old clock reading at the instant when the constraint was first established). The items in a rtdb age with time (Kopetz, 1989a). Hence, an important form of timing constraint is a duration associated with (attribute, value) pairs of each data item. Let ($A_X$, $A_X$'value) be a pair associated with a data item x in the rtdb, and let $T_{AX}$ be a rigid variable which stores the reading of the external clock in the Constrainer state when ($A_X$, $A_X$'value) was last modified. Let *pname* identify a timing constraint on $A_X$'value. The timing constraint on ($A_X$, $A_X$'value) is expressed as constraints in the rtcb with predicates of the form pname(($A_X$, $A_X$'value), $\delta$), which asserts that ($A_X$, $A_X$'value) has a lifespan of $T_{AX} + \delta$ ticks of the external clock.

*Decomposition of a Temporal Formula*

The AcceptConstraint ( criterion, know, q.time ) rule checks the validity of a constraint expressed as a $TL_{rt}$ formula. This entails the decomposition of a constraint, which is a temporal formula. A temporal formula specifies a finite, directed, labeled graph. Each node in the graph is labeled with a non-temporal predicate which is evaluated with respect to data item values. Using a technique similar to the one suggested by (Wolper, 1981; Lipeck, 1987), each of the following temporal formulas is decomposed into a non temporal and a temporal component. This technique constructs a directed graph. Let P be a temporal formula, then

$\Box P \equiv seq(P, \Box P)$ --graph with nodes P, $\Box$ P

$\Diamond P \equiv true\ U\ P$ --graph with nodes true,...,P

$seq ( P_1, P_2, ...P_n ) \equiv P_1$ before $seq ( P_2, ...P_n )$

--graph with nodes $P_1$, $seq(P_2,...)$

This decomposition technique is utilized by the Constrainer. The state variable q.Pred stores the non-temporal component of a constraint. The decomposition of a real-time, temporal formula is performed by the Decompose subrule:

```
Decompose ( q | tail, formula ) :-
   q.Pred is formula.
Decompose ( q | tail,   " seq ( P | tail ) " ) :-
Decompose ( q ,  P ),
Decompose ( tail,  " seq (tail) ").
Decompose ( q | tail,  " □ " | formula ) :-
   q.Pred is formula,
Decompose ( tail,  formula ).
Decompose ( q | tail,  " ◇ " | formula ) :-
   not( ( not Decompose( q | tail,  formula )),
      q is " ")).
```

The reason for decomposing and storing the non-temporal portion of a constraint in q.Pred is one of efficiency. Since, q.Pred would possibly be checked subsequently for consistency with respect to other constraints, it would be inefficient to decompose the original temporal constraint each time along with other constraints in the rtcb constraint base.

*Historyless Integrity checking*

The Constrainer provides historyless integrity checking. That is, the Constrainer does not rely on a state history. The only state record it relies on is the record for the current state. Instead, past values of selected state variables are stored in rigid variables such as $T_{Ax}$, which saves the time of the last modification of a data item x with attribute $A_x$ in the rtdb. This form of historyless integrity checking is in terms of the future fragment, which improves on the minimal history checking in Lipeck (1987, 1990), intended for conventional rather than real-time databases. The historyless checking of the Constrainer also contrasts with the form of historyless integrity checking in Chomicki (1990), which is in terms of the past fragment of temporal logic and which is designed for conventional, non-real-time databases.

## CONSISTENCY, INTEGRITY, AND KNOWLEDGEABILITY ISSUES

This section examines the consistency issues relative to a rtcb, rtdb, and design of a controller. As will be shown, it is possible to incorporate what is best called a knowledge-discipline in the design of the controller. This knowledge-discipline requires a controller to limit its usage of rtdb values only to those values which satisfy each of their constraints in the rtcb.

*Consistency of rtcb constraints*

The Constrainer guarantees that every new constraint added to the rtcb is consistent with existing constraints. Let p and q be constraints, then p and q are consistent if $q \neq \neg p$.

> Fact $1.1^{o}$ (Asking) Let p be a constraint to be added to the rtcb. The IsConsistent subrule of the Constrainer returns Yes if p is consistent with each constraint in the rtcb; otherwise, the IsConsistent subrule returns a No.
> Fact $1.2^{o}$ (Telling) The Tell operation of the Constrainer succeeds if Fact $1.1^{o}$ holds.

The IsConsistent subrule (see Rule IsC) is used by the Constrainer to check the consistency of new constraints relative to existing constraints in the rtcb and is given as follows:

```
IsConsistent ( know | klist, constraint,  q.time, q.Pred ) :-
  CheckTime ( know, constraint, q.time ),
  CheckCriterion (know, constraint ),
  CheckFormula (know, constraint, q.Pred),
  IsConsistent ( klist, constraint, q.time, q.Pred ).
```

The subrule CheckTime (see Appendix B) checks the compatibility of the duration of the new constraint with each of the constraints in the rtcb relative to the current time. The subrule CheckCriterion ensures that both the sentences are about the same data item. The subrule CheckFormula looks for contradiction between a new constraint and an existing constraint. From Facts $1.1^{o}$ and $1.2^{o}$, we know that if the IsConsistent subrule returns Yes, the Constrainer adds the constraint p to the rtcb. Otherwise, in the case where IsConsistent returns No, the Constrainer rejects the constraint p. Hence

Proposition $1^{o}$ (Consistency of rtcb).
Each constraint added to the rtcb is consistent with every other constraint in the rtcb at the time it is added.

The maintenance of the rtcb relative to timing constraints which have expired, is not addressed in this article (expired timing constraints are ignored). Proposition $1^{o}$ points to the fact that all current timing constraints in the rtcb are consistent with each other.

*Integrity of rtdb Items*

The rtcb provides a set of dynamic constraints associated with each rtdb data item. A *dynamic constraint* on a data item x in an rtdb constrains values of x over a sequence of rtdb states. Let $(A_X, A_X\text{'value}) \in x$, and let $(A_X, A_X\text{'value})$'constraint be a set of constraints on $(A_X, A_X\text{'value})$ in rtcb. Every item in the real-time database has at least one rtcb constraint on it. This fact is expressed formally in Fact $2^{o}$.

Fact $2^{o}$ (minimum constraints)
□ ( $\forall$ $x \in$ rtdb, $(A_X, A_X\text{'value}) \in x$ | $(A_X, A_X\text{'value})$' constraint | >= 1 )

Let $q_o$ be the initial rtdb state, $q_o$' be the initial rtcb state, $q_o$" be the initial state of a real-time controller which will rely on the use of the rtdb in controlling the behavior of a plant, and the initial real-time system state be $Q_o = (q_o, q_o', q_o")$. Let $A_X$'value be the value of the attribute $A_X$ of data item x in the rtdb and let q".action be the name of the action being performed by the controller in state q". To express the *satisfaction of a predicate* P in a state q, we write sat(q, P). Before any item in the rtdb is instantiated (i.e., assigned any value), at least one constraint associated with that item must be added to the rtcb. This last assumption is expressed formally in Fact $3^{o}$ relative to the *initial* integrity system state $Q_o$.

Fact $3^{o}$ (constraintbase initialization)
( $\forall$ $x \in$ rtdb, $(A_X, A_X\text{'value}) \in x$, $(A_X, A_X\text{'value})$'constraint $\in$ rtcb:
sat( Constrainer ( $Q_o$, Tell, , , $(A_X, A_X\text{'value})$'constraint )
and $(A_X, A_X\text{'value})$ = ( , ) and q".action = idle)

This anchors the integrity system in state $Q_0$. From Facts $2^0$ and $3^0$, we know that every (attribute, value) pair ($A_X$, $A_X$'value) of each item x in the rtdb is constrained with a timing constraint and/or other forms of constraints. In the case where each ($A_X$, $A_X$'value) has a timing constraint using Def. $0.2^0$, we can prove

Proposition $2^0$
If there is a ($A_X$, $A_X$'value)'constraint for every x in the rtdb, and $\exists$ c $\in$ ($A_X$, $A_X$'value)'constraint which is a timing constraint, then the rtdb is temporally complete.

All instantiations of data items are made by the Constrainer, which is invoked either by the controller or by a human operator. This idea is expressed in Fact $4^0$ .

Fact $4^0$ (item addition)
( $\forall$ x $\in$ rtdb : sat( Constrainer ( q, Add, $A_X$, $A_X$'value, ) = Yes
and rtdb'OUT=rtdb'IN $\cup$ { ($A_X$, $A_X$'value) }). --side effect

If the $A_X$'value to be added to the database satisfies all the constraints in ($A_X$, $A_X$'value)'constraint, then the Constrainer returns a "yes" and the ($A_X$, $A_X$'value) is added to the rtdb, otherwise the Constrainer returns a "no" and the new value is rejected. There is a similar requirement for all updates to data items in the rtdb expressed by Fact $5^0$

Fact $5^0$ (item updation)
( $\forall$ x $\in$ rtdb : sat( Constrainer(q , Update, $A_X$, $A_X$'value, ) = Yes
and rtdb'OUT = rtdb'IN $\cup$ {$A_X$'value / $A_X$'oldvalue } ). --side effect

Facts $4^0$ and $5^0$ say that every instantiation of a constrained data item requires a "yes" answer from the Constrainer. We define *a data item x in the rtdb has integrity* in the current system state Q as follows:

Def. $1^0$ (data integrity)
Let q be a state of the Constrainer.

HasIntegrity ( q, x ) $\equiv \forall$ $A_X$ : Constrainer ( q, Verify, $A_X$, , ) = Yes.

The only modifications to the database is through an update or an add operation, since the Constrainer is the sole manager of the rtdb. By appealing to Facts $4^o$ and $5^o$, and examining the actions of the update and add operations of the Constrainer, the following propositions can be proved constructively:

Proposition $3^o$ ( Constrainer Update )
From $\forall A_X$ : Constrainer (q, Update, $A_X$, $A_X$'value, ) infer HasIntegrity ( q, x ) .

Proposition $4^o$ ( Constrainer Addition)
From $\forall A_X$ : Constrainer ( q , Add, $A_X$, $A_X$'value, ) infer HasIntegrity ( q, x ) .

Facts $1^o$ through $5^o$ and Props. $3^o$ and $4^o$ make it possible to prove that every data item in the rtdb has integrity in all database states.

Proposition $5^o$ (Integrity)
From $\forall A_X$ , $x \in$ rtdb: $q_0,..., q_i \models (A_X, A_X\text{'value})\text{'constraint}$
infer $q_0,..., q_i \models$ HasIntegrity($q_\alpha$ / q, x), $\alpha$ in ($q_0,..., q_i$ ).

*Knowledgeably Complete Controller*

A key issue in designing a controller for a real-time system is the extent to which accesses by the controller to a real-time database are constrained.

Fact $5^o$ (access)
An access to the rtdb is either through either a read, an Add, or Update of the Constrainer.

From Fact $5^o$, we can prove

Proposition $6^o$ (access)
$\forall$ ($A_X$, $A_X$'value)'constraint in rtcb, sat( q, ($A_X$, $A_X$'value)'constraint) infer
Constrainer (q, Update, $A_X$, $A_X$'value, )
or Constrainer ( q , Add, $A_X$, $A_X$'value, )
or Constrainer ( q , Read, $A_X$, $A_X$'value, )

This design issue leads to the characterization of a controller in Def. $2^o$.

Def. 2° A *knowledgeable controller* is a controller in which at least one of its accesses to a constrained item x ∈ rtdb is a result of determining that x satisfies all of its constraints.

That is, a controller is knowledgeable if it Asks the Constrainer about the integrity of needed information in the rtdb and/or it relies on the Constrainer to read a data item in the rtdb. From Def. 2° and an analysis of the read operation for the Constrainer, we can prove

Proposition 7° If a controller invokes the Read operation of the Constrainer when it wishes to access a data item in the rtdb, then the controller is knowledgeable.

To guarantee that every data item x in rtdb accessed by a controller has integrity, it is necessary to enforce the discipline in the design of the controller reflected in Def. 3°.

Def. 3° A controller is *completely knowledgeable* if each of its accesses to an item x in the rtdb is a result of determining that x satisfies all of its constraints.

In the case where each controller access to the rtdb is through the Constrainer, then we know from Def. 3° and Propositions 6° and 7° that the following Proposition holds:

Proposition 8° If all accesses of a controller are *through the Constrainer*, then the controller is completely knowledgeable.

In summary, we have formally defined consistent rtcb, a basis for the integrity of items in the rtdb, and the notion of a knowledgeably complete controller.

## EXAMPLE CONTROLLER

This section illustrates an explicit clock, temporal specification of a knowledgeable controller, rtdb, rtcb in terms of a variation of the gasburner control system described by Nordahl (1989). The purpose of the controller is to guarantee safe operation of the gas burner.

### *Description of Behavior of the Controller for the Burner*

The components of a controller for the gas burner are shown in Fig. 2. The controller shown in Fig. 2 starts by turning on the CU (control unit), which turns the pilot (small gas source). Then the CU fires the spark plug, which ignites the gas coming from the pilot. Once the flame from the ignition of the pilot gas by the spark plug has started (the mirror sends a message to the CU that the flame is on), then the CU turns the main gas line on. The ignition of the gas from the main prompts the CU to turn the blower on (the blower supplies oxygen for combustion, blows the heat out of the combustion chamber to the object to be heated, and evacuates carbon monoxide from the combustion chamber). A temporal specification of the controller behavior written in $TL_{rt}$ is given in Fig. 3. The specification in Fig. 3 gives the initialization checks performed by the controller to ascertain that the individual plant components are usable. This behavior is specified as a temporal formula which asserts that infinitely often ( $\Diamond\omega$ ) the controller performs a sequence of operations. In Figure 3, the controller repeatedly consults the Constrainer concerning the appropriate sequence of timed settings for the safe operation of the gas burner. This specification also refers to another rtdb entity called Unit, which stores the settings (on, off) for all of the hardware being controlled (main, pilot, CU, spark, and so on). The notation CU: Unit.setting indicates that the CU reads the setting of Unit (in the rtdb) and sends this information to the hardware identified by Unit. The controller in Figure 2 depends on the Constrainer to determine acceptable control values to communicate to the hardware. For example, the controller Asks the Constrainer if the CU is usable before it turns the CU on. Hence, from Def. $2^{o}$, we know that the controller for the gas burner is knowledgeable.

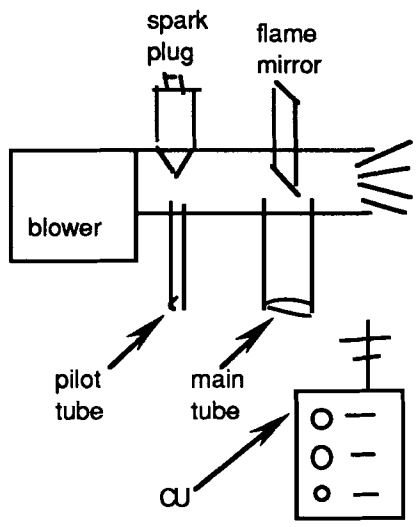


Fig. 2 Diagram of Gas Burner

$\Diamond\omega$ seq ( start,

Constrainer($q_0$,Ask,CU.name,H110) = Yes $\Longrightarrow$ CU: H110.on

**or**

seq( Constrainer($q_0$, Ask,CU.name, H110) = No $\Longrightarrow$
Constrainer($q_1$,Tell, CU.name = H115, IsValid(H115, 100)),
Constrainer($q_2$, Update, Unit.id | Unit.setting, H115 | on) ),

Constrainer($q_3$, Ask, Pilot.name,a) = Yes $\Longrightarrow$ CU: a.on,

Constrainer($q_4$, Ask, Spark.id, JU13) =Yes $\Longrightarrow$ CU: JU13.on; delay(3 sec),

(Constrainer($q_7$, Ask, Main.name, X) = Yes
and Unit.name = flame and Unit.setting = on) $\Longrightarrow$ CU: X.on , ..., CU: X.off)

Fig. 3 Temporal Specification of Controller for Gas Burner

*Real-Time Database for Gas Burner Controller*

Let gb = gas_burner and $rtdb_{gb}$ be the real-time database for the controller for the gas burner (see Table 1). Let Spark, Main, Pilot, CU, and Unit be names of entities in the $rtdb_{gb}$. Then $rtdb_{gb}$ is defined as follows:

$rtdb_{gb}$ = {Spark, Main, Pilot, CU, Unit}

where each entity is specified as a set of (attribute, value) pairs, which define the rows of the rtdb entity table.

TABLE 1. REAL-TIME DATABASE FOR GAS BURNER CONTROLLER

```
Spark =  { ( (id, JU13),  (upper, 500), (lower, 450)),
          ((id, JU15), (upper, 600), (lower, 350))}

Main =  {   ( (name, X), (gas_flow, g1)),
          ( (name, Y), (gas_flow, g2))}

Pilot = { ( (name, a), (capacity, c1)),
          ( (name, b), (capacity, c2))}

CU =  { ( (name, H110), (brand, HW)),
          ( (name, H115), (brand, HW))}

Unit = { ((name, Power),(setting,on)),
          ((name, X),(setting,on)),
          ((name, Y),(setting,off)),
          ((name, a),(setting,on)),
          ((name, b),(setting,off)),
          ((name, JU13),(setting,on)),
          ((name, JU15),(setting,off)),
          ((name, Blower),(setting,on)),
          ((name, CU),(setting,on)),
          ((name, ignition),(setting,on)),
          ((name, flame),(setting,on)),

       }
```

*Constraint Base for Gas Burner Controller*

Let $rtcb_{gb}$ be the real-time constraint base for the controller for the gas burner (see Table 2). The constraints in the $rtcb_{gb}$ have time limits placed on them (i.e., these constraints have time limits on their validity). For example, the constraint (in Table 2) with selection criterion CU.name = H110 has an upper bound of 300 ticks after its installation (point of observation) in the constraint base. For convenience, the name of the attribute has been omitted from column 2 of Table 2. The timing constraint (i.e. duration for the entire constraint) indicates the lifespan of the constraint. After the expiration of a timing constraint, it must be refreshed (reset, possibly modified) before the constraint can be used by the controller. In addition, there is a timing constraint on the attribute itself of the rtcb. For example, CU.brand = HW has an upper time bound of 50 ticks after its last modification. This value is its t_val (point of validity). That is, after 50 ticks, the

value for CU.brand will no longer be valid. There are some timing constraints in Table 2 with no upper bound on the duration of their validity (i.e., duration for the constraint = ∞ ).

TABLE 2. REAL-TIME CONSTRAINT BASE FOR GAS BURNER CONTROLLER

| Clock Ticks | Select Criterion | Constraint |
|---|---|---|
| 300 | H110 | IsValid(CU.brand = HW, 50) |
| 250 | H115 | IsValid(CU.brand = HW, 30) |
| 110 | X | IsValid(Main.name = X, 40) |
| 90 | Y | IsValid(Main.name = Y, 80) |
| 9 | a | IsValid(Pilot.name = a, 2) |
| 19 | b | IsValid(Pilot.name = b, 5) |
| 50 | JU13 | IsValid(Spark.id = JU13, 30) |
| 60 | JU15 | IsValid(Spark.id = JU15, 20) |
| ∞ | a and JU13 and ignition and flame | seq(Ison(a) and IsOn(JU13), IsOn(ignition, δ), IsOn(flame, δ')) |
| ∞ | X and flame and blower | □ (IsOff(X) and IsOff(flame) before IsOn(blower, δ). |
| ∞ | X and a and flame | □ (IsOff(X) and IsOff(a) before IsOff(flame, δ') |

In addition to timing constraints on the attributes, the last three rows of Table 2 provide examples of constraints which specify a precise temporal ordering of the changes to values of the real-time database. Such constraints are specified as temporal predicates (see Appendix A). For example, the rtcb constraint on the gas burner ignition sequence is given as

delay(∞)• Unit.name = a and Unit.name=JU13 and Unit.name = ignition
Unit.name = flame •
seq(Ison(a) and IsOn(JU13), IsOn(ignition, δ), IsOn(flame, δ'))

The timing component delay(∞) of this constraint asserts that this constraint is not refreshed (i.e., there no upper bound on the duration of their validity). The selection criteria specifies all of the Unit attributes referenced by this constraint. The assert portion of this constraint specifies an ordering on the evaluation of predicates where

(Pilot a is turned on (IsOn(a))
*and* Spark Plug JU13 is turned on (IsOn(JU13)))
*before* ignition is turned on
within $\delta$ ticks of the external clock (IsOn(ignition, $\delta$))
*before* flame is on within $\delta'$ ticks (IsOn(flame, $\delta'$))

This rtcb constraint constrains rtdb values over a sequence of rtdb states. The realization of such a constraint is made possible by an interaction of the controller and the Constrainer in instantiating the rtdb items.

## CONCLUSION

A knowledge-based approach to overseeing the timing as well as other constraints on items in a real-time database ensures the integrity of the information. In addition to making a provision for timing constraints on items in the rtdb, the well-formed presumptions of the real-time constraint base can also have timing constraints. Explicit clock temporal logic is useful in writing hard, real-time constraints on items in the rtdb, presumptions in the rtcb, and on the behavior of a controller. All three forms of timing constraints have been illustrated in this paper. Finally, the foundation for establishing a temporally complete rtdb and knowledgeably complete controller has been presented.

## ACKNOWLEDGEMENT

We would like to thank Gideon Frieder and the other members of the School of Computer & Information Science at Syracuse University for providing an excellent environment for this research.

## REFERENCES

Alur, R., and D. Dill (1990). Automata for modeling real-time systems. *Automata, Languages and Programming.* Lecture Notes in Computer Science 443, Springer Verlag, NY. Pp. 322-335.

Alpern, B.L. (1986). *Proving Temporal Properties of Concurrent Programs: A Non-Temporal Approach.* Ph.D. dissertation, Cornell University, TR-86-732.

Chomicki, J. (1990). History-less Checking of Dynamic Integrity Constraints. Report TR-CS-90-19, Dept. of Computing & Information Sciences, Kansas State University.

Cronhjort, B. (1988). Specification and Quality Assurance of Real-Time Software. *IFAC/IFIP Symposium on Software for Computer Control*, pp. 9-14.

Guessoum, A. and Lloyd, J.W. (1990). Updating Knowledge Bases. *New Generation Computing*, vol. 8, pp. 71-89.

DoD (1983) U.S. Dept. of Defense, *Reference Manual for Ada Programming Language,* ANSI/MIL STD 1815A-1983. Springer-Verlag, NY.

Handelmann, D.A. and Stengel, R.F. (1988). Perspectives on the Use of Rule-Based Control. *Proceedings of the IFAC Workshop on Artificial Intelligence in Real-Time Control*, pp. 27-32.

Hankley, W., and J.F. Peters (1990a). Temporal Specification of Ada Tasking. *Proceedings of the 23rd Hawaii International Conference on System Sciences: Software Track*, Vol. II, pp. 410-419.

Hankley, W., and J.F. Peters (1990b). A Proof Method for Ada/TL. *Proceedings of the 8th ACM-IEEE Annual National Conference on Ada Technology*, pp. 392-399.

Harel, E., et al (1990). Explicit Clock Temporal Logic. *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pp. 402-413.

Henzinger, T.A., et al (1991a). Temporal Proof Methodologies for Real-Time Systems. *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pp. 353-366.

Henzinger, T.A. (1991b). The Temporal Specification and Verification of Real-Time Systems. Ph.D. Dissertation, Computer Science Department, Stanford University, 1991.

Kopetz, H., and K. Kim (1988). Consistency constraints in distributed real time systems. *Proceedings of the 8th IFAC Workshop on Distributed Computer Control Systems*, pp. 29-34.

Kopetz, H. (1989a). Real-Time Systems. Research Report Nr. 12/89, Institut fur Technische Informatik, Technische Universitat, Wien, Austria.

Kopetz, H. (1989b). Design of Real-Time Computing System. *Proceedings of Joint Univ. of Newcastle Upon Tyne/International Computers Limited Seminar*, pp. IV.25-IV.58.

Kröger, F. (1985). Temporal Logic of Programs Lecture Notes. Report TUM-I8521, Institut fur Informatik, Technische Universitat Munchen.

Levesque, H.J. (1981). *A Formal Treatment of Incomplete Knowledge Bases.* Ph.D. Dissertation, Department of Computer Science, University of Toronto.

Lipeck, U. (1987). Monitoring dynamic integrity constraints based on temporal logic. *Information Systems*, vol. 12, no. 3, pp. 255-269.

Lipeck, U. (1990). Transformation of Dynamic Integrity Constraints into Transaction Specifications. *Theoretical Computer Science*, vol. 76, no. 1, pp. 115-142.

Manna, Z. and Pnueli, A. (1981). Verification of Concurrent Programs, Part 1: The Temporal Framework. Report No. STAN-CS-81-836, Dept. of Computer Science, Stanford University.

Manna, Z. and Pnueli, A. (1983). Verification of concurrent programs: a temporal proof system. Technical Report, Dept. of Computer Science, Stanford University, June, 1983.

Nordahl, J. (1989). A Real-Time Temporal Logic Specification of a Safety Critical System. ID/DtH JNo2, Dept. of Computer Science, Technical University of Denmark.

Ostroff, J.S. (1989). *Temporal Logic for Real-Time Systems*, John Wiley & Sons, Inc., New York.

Ostroff, J.S. and Wonham, W.M. (1990). A Framework for Real-Time Discrete Event Control. *IEEE Transactions on Automatic Control*, vol. 35, no. 4, pp. 386-396.

Peters, J.F. (1990a) *Constructive Specification of Communicating Processes Using Temporal Logic.* Ph.D. dissertation, Computing & Information Sciences, Kansas State University.

Peters, J.F., and W. Hankley (1990b). Proving Specifications of Tasking Systems Using Ada/TL. *Proceedings of ACM Tri-Ada'90*, pp. 4-13.

Peters, J.F. and Ramanna, S. (1991a). Modelling Timed-Behavior of Real-Time Systems with Temporal Logic. To appear in *Cybernetics and Systems: An International Journal.*

Peters, J.F. and Ramanna, S. (1991b). Constructing Real-Time Systems from Temporal I/O Automata. Report No. SU-CIS-91-22, School of Computer & Information Science, Syracuse University.

Peters, J.F. (1991c). Prototyping Provably Correct Real-Time Systems. Report No. SU-CIS-91-23, School of Computer & Information Science, Syracuse University.

Peters, J.F. (1991d). RTS Prototyper's Workbench: A Tool for Rapid Prototyping Provably Correct Real-Time Systems. Research Report, Research & Development Laboratories, Culver City, CA, U.S.A.

Ramanna, S. (1990a). *Temporal Logic in the Design of Integrity Systems.* Ph.D. Dissertation, Dept. of Computing & Information Sciences, Kansas State University.

Ramanna, S. et al. (1990b). Designing a Dynamic Integrity Constraint Checker with Nonmonotonic Logic. *Proceedings of the 14th Annual International Computer Software & Applications Conference* (COMPSAC-90), Chicago, Illinois.

Ramanna, S. and Peters, J.F. (1991) Explicit Clock Logic in Timing Constraints for Real-Time Systems. Report No. SU-CIS-91-26, School of Computer & Information Science, Syracuse University.

Wolper, P. and Manna, Z. (1981). Synthesis of Communicating Processes from Temporal Logic Specifications. LNCS 131. Springer-Verlag, N.Y. Pp. 253-281.

## APPENDIX A

Note: the italicized terms in the following grammar are taken directly from Ada (DoD 1983).

| | | |
|---|---|---|
| wfc | ::= | [delay( const )] • SelectionCriteria • ( assert ) |
| const | ::= | real |
| SelectionCriteria | ::= | Attribute *relational_operator* Value {*logical_operator* SelectionCriterion} |
| Attribute | ::= | *identifier* |
| Value | ::= | *numerical_literal* \| *character_literal* |

| | | |
|---|---|---|
| | \| | *string_literal* |
| assert | ::= | Temporal_Predicate |
| TemporalPredicate | ::= | UTempOp [ ( ] predicate [ ) ] |
| | \| | SeqPredicate |
| predicate | ::= | ∀ x.P \| ∃ x. P |
| | \| | q and r \| q or r \| ¬ p \| q ⟹ r |
| | \| | q BTempOp r |
| | \| | P(T [, T]), where n >= 0 |
| T | ::= | identifier |
| | \| | F(T [, T]), where n >= 0 |
| SeqPredicate | | seq ( PredList ) |
| PredList | ::= | predicate [, PredList ] |
| UTempOp | ::= | □ \| ◇ |
| BTempOp | ::= | U \| before |

## APPENDIX B

```
/*   Extract interval, criterion, formula  */

Extract(constraint,delay,criterion,formula)  :-
  ExtractTime ( constraint,  delay ),
  ExtractCriterion ( constraint,  criterion ),
  ExtractFormula ( constraint,  formula ).
ExtractTime  ( constraint,  delay ) :-
  FrontToken( constraint,  first, next ),
  FrontToken( first,  delay , next ).
ExtractCriterion ( constraint,  criterion ) :-
  FrontToken(constraint,  first, second ),
  FrontToken( second,  criterion , next ).
ExtractFormula  ( constraint,  formula ) :-
  FrontToken( constraint,  first, second ),
  FrontToken( second,  third , [] ).


/*  consistency checking  */

CheckTime (know, constraint, q.time ) :-
  ExtractTime  ( know,  oldtime ),
  ExtractTime  (constraint,  newtime ),
  CompareTime ( oldtime, newtime ).
CheckCriterion ( know, constraint) :-
  ExtractCriterion  ( know,  oldcriterion ),
  ExtractCriterion ( constraint, newcriterion ),
  CompareCriterion(oldcriterion,newcriterion).
CheckFormula (know, constraint, q.Pred ) :-
  ExtractFormula  ( know,  oformula ),
  ExtractFormula  ( constraint,  nformula ),
  CompareOp (oformula, nformula ),
  not IsRefuted( oformula, nformula, q.Pred ).
CompareTime ( oldtime, newtime) :-
  oldtime <= newtime.
```

```
CompareCriterion(oldcriterion,newcriterion)  :-
  oldcriterion =  newcriterion.
CompareOp(" □ " | oformula," □ " | nformula) .
CompareOp(" ◇ " | oformula," ◇ " | nformula ).
CompareOp("seq" | oformula,"seq" | nformula).
CompareOp(" □ "| oformula,"seq" | nformula):-
CompareOp( " □ " | oformula, op | nformula ).
CompareOp("seq" | oformula," □  " | nformula ).
CompareOp ( op | nformula,  " □ " | oformula ).
IsRefuted ( oformula, nformula, q.Pred ) :-
   split( oformula, oldnontemp ),
    oldnontemp and not  q.Pred.


/*  admissibility of an operation */


Admissible ( q, op, attrib, formula, values ) :-
  substitute ( attrib, formula, values, eval ).


/* formula substitution */


substitute ( sub, term, sub1, term1 ) :-
  term = .. [ F | Args ],
  sublist ( sub, Args, sub1, Args1 ),
  term1 = .. [ F | Args1 ].
sublist ( _, [], _, [] ).
sublist ( sub, [term| terms], sub1, [term1| terms1] ) :-
  substitute ( sub, term, sub1, term1 ) ,
  sublist ( sub,  terms, sub1,  terms1 ).
  eval is term1.


/*  accept   criterion */


AcceptCriterion ( criterion, db ) :-
  collect_items  ( criterion, itemlist ),
  member ( itemlist, db ).
member ( item | itemlist,  item | db ) :-          /* item found */
  member ( itemlist, db ).
member ( item | itemlist,  data | db ) :-          /* item not found */
  member ( item | itemlist, db ).


/*  find constraint set */


FindSentence (criterion, know| q.cb, knowset):-
  ExtractCriterion  ( know,  newcriterion ),
  CompareCriterion ( criterion, newcriterion ),
  FindSentence(criterion, q.cb, know | knowset);          /* match found */
  FindSentence ( criterion,  q.cb, knowset )              /* match not found  */
```